HZ BOOKS
华章科技

资深软件开发专家Josh Carter 20余年编程生涯心得体会，从程序员成长视角，系统总结和阐述了专业程序员在编程技能和方法、编程工具、自我管理、团队协作、组织架构、工作态度和原则、自我学习和持续改善等方面应该掌握的33个技巧。

华章程序员书库

The Pragmatic Programmers

New Programmer's Survival Manual
Navigate Your Workplace, Cube Farm, or Startup

程序员修炼之道

专业程序员必知的33个技巧

（美） Josh Carter 著
胡键 译

机械工业出版社
China Machine Press

华章程序员书库

程序员修炼之道：专业程序员必知的33个技巧

New Programmer's Survival Manual

Navigate Your Workplace, Cube Farm, or Startup

（美）Josh Carter 著

胡 键 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

程序员修炼之道：专业程序员必知的 33 个技巧 /（美）卡特（Carter, J.）著；胡键译 . —北京：机械工业出版社，2013.1
（华章程序员书库）
书名原文：New Programmer's Survival Manual: Navigate Your Workplace, Cube Farm, or Startup

ISBN 978-7-111-41164-2

Ⅰ . 程… Ⅱ . ① 卡… ② 胡… Ⅲ . 程序设计 – 基本知识 Ⅳ . TP311.1

中国版本图书馆 CIP 数据核字（2013）第 006745 号

本书版权登记号：图字：01-2012-6853

这是每一位致力于成为专业程序员的软件开发新手都应该阅读的一本书。它是资深软件开发专家 Josh Carter 20 余年编程生涯的心得体会，从程序员成长的视角，系统总结和阐述了专业程序员在专业技能、编程工具、自我管理、团队协作、工作态度以及需要采取的行动等方面应该掌握的 33 个非常重要且实用的技巧。作者以自己以及身边的同事积累下来的经验、犯过的错误为素材，旨在为新人们引路，让他们在能力修炼的过程中少走弯路！

全书分为四个部分：第一部分（技巧 1~14），从编程技能和工具使用两个方面总结了 14 个技巧，包含如何正确地书写代码、测试驱动设计、管理代码复杂度、改善遗留代码、代码评审、开发环境优化、自动化等；第二部分（技巧 15~24），从自我管理和团队协作两个方面总结了 10 个技巧，包括如何树立自我形象、压力管理、建立良好人脉和高效会议等；第三部分（技巧 25~30），介绍了典型高科技公司的组织结构以及你在整个公司中的位置，并且阐述了薪酬分配的问题；第四部分（技巧 31~33），介绍了在日常工作中如何持续改善自己的工作和学习状态。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：秦　健
　　印刷
2013 年 1 月第 1 版第 1 次印刷
186mm×240mm • 13.25 印张
标准书号：ISBN 978-7-111-41164-2
定价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

译 者 序

说实话，在翻译完《The Cloud at Your Service》[⊖]之后，我本打算暂时不再接手翻译工作了。一方面是因为工作越来越忙，另一方面则是因为翻译实在是一件费神费力的事情。可当机械工业出版社的编辑把这本书摆在我面前时，我还是背弃了自己想要休息的想法。

我注意这本书其实有一段时间了，因为这是一本独特的书。虽然面向程序员，却并非一本单纯的技术书籍。尽管英文书名包含 New Programmer，但对于入行多年的“老兵”同样能做到开卷有益。事实上，在翻译本书的过程中，我本人也从中获益匪浅。

对于刚踏入社会工作的毕业生，要完成从学生到社会人的转变并非易事。尤其是那些从事编程工作的社会新鲜人，有许多的观念和事情需要去转变和学习。比如：你以前可能孤身一人开发完成整个应用，现在则可能要跟他人合作；你之前的代码可能是自己分目录保存不同副本，如今可能要用到类似 Git 这样的版本控制软件；你之前的代码可能是自己手动测试，而新公司可能要求你写自动化的单元测试……除了这些纯技术上的变化外，你还会面临一些诸如绩效考核、配合市场人员宣传、职业生涯这样的新事物。

所有以上内容，你都会在本书中找到相应的内容和建议。虽然读一本书不会让你马上转变身份，但起码会给你提供帮助，让你感觉不再孤单。说句心里话，要是当年我有一本这样的书该有多好！

虽然时光无法倒流，但现在也为时未晚。对于在踏入编程大门初期错过了本书的朋友们，同样能在本书中找到适合自己的内容。你是否每到绩效考核时就头痛？你是否觉得长时间坐在办公室内编程损害了你的健康？你是否讨厌开会，认为它们纯粹是浪费时间？你是否关心所在公司的营业情况？你有没有想过换个工作？……打开这本书，读一读别人的想法和做法。

除了本书的内容外，我接下这本书的翻译其实还有一个私心：希望我的同行妻子也能从中汲取些营养。于是，在“初译—审校—润色—审校”的过程中，她最终承认这是一本好

⊖ 本书中文版《云计算揭秘——企业实施云计算的核心问题》已由机械工业出版社引进出版，ISBN: 978-7-111-38494-6。——编者注

书，决定要将从书中学到的经验用到实际中——这是“润物细无声”的典型案例。

最后，我要感谢参与本书的所有的机械工业出版社的编辑，让我有机会负责本书的翻译工作，而且有一个非常愉快的合作体验；同时，我还要感谢参与本书审校工作的朋友：朱晓弟、仵建锋和焦斌。当然也不能遗漏老婆，不仅是因为你的审校次数最多、最深入，而且还因为你对于我生活的支持和帮助！

胡　键

于西安

前　言

今天是上班的第一天。你拿到了编程执照，找到了工作，坐在你的工作站旁……下面该干什么？在你面前，一座新的丛林正等着你：

- 按行业规模编程，其中的代码库规模以上千（或几十万）行代码来衡量。你怎样才能快速入门，开始作出贡献？
- 遨游在除了程序员之外还有许许多多其他角色的组织内。当要了解产品特性时，你向谁请教？
- 每年都有所成就。当绩效考评潜伏在地平线上时，你知道老板的目标吗？你知道自己将如何被判定吗？

还有很多很多。你的编程技能只是工作第一年里要用到的技能的一部分。

我们中的幸运儿会有识途老马充当向导。本书则是一位虚拟向导，它将为你指明方向，指出前方的高山和峡谷，同样也将让你避免跌入令人讨厌的陷阱。

我的经历

你或许能从我在 1995 年上大学时的情景中找到一些与自己经历相似之处：我一开始走的是传统老路，一名杜克大学计算机科学与电子工程系的学生。我曾找过我的导师，询问哪些课程最有利于我未来求职。他是个聪明的家伙——一名罗德学者[⊖]和这间工程学校冉冉升起的新星——他的回答是：“我不知道。我从未在行业里工作过一天。”

我大失所望。我想构建真实、有人买的产品，而不是写研究论文。因此，那个夏天我设法加入了硅谷一家方兴未艾的创业公司：General Magic。它是由当初创造了 Macintosh 计算机的同一拨人（Andy Hertzfeld 和 Bill Atkinson）创建的。我的同事包括来自苹果公司 System 7（操作系统）团队的一些顶级开发者和后来创建了 eBay 的那个哥们儿。

⊖ 罗德奖学金得主的称号。罗德奖学金由塞西尔·罗德斯于 1902 设立，已有超过百年历史，它是世界级的奖学金，有“全球本科生诺贝尔奖”之称的美誉。（摘自维基百科）——译者注

我在两个月的实习期内学到的编程知识比我在学校里两年学到的还要多。我给杜克大学打了个电话，说我不打算回学校了。就这样，我在行业里的狂野冒险开始了。

现在说说你

本书的读者可大致划为如下几类：

- 选修计算机科学和有这样疑问的大学生和将要毕业的学生：“现实世界里的编程是这个样子吗？”（简单说：不是。）
- 具有其他背景，因为爱好或副业而涉足编程，现在想将其作为全职工作的职业人士。
- 正在考虑编程行当，但想找些书中和课堂上没有教过的东西的其他人。

不论你属于哪种类型，你现在的情况是：到了靠编码为生的时候。就代码部分而已，市面上以之为主题的书可谓汗牛充栋。但讨论跟这个工作相关的其他方方面面的书籍，就不见得有那么多了——这正是本书的初衷。

对于转行的从业者，有些章节可能对你没多大用处——假如你具有市场营销的背景，那就用不着我来告诉你市场营销究竟为何物。但你还是可以从工程部门的运作方式以及代码从概念到发布的演变过程的相关内容中获益。

本书组织结构

本书以技巧的形式写就，每个技巧用寥寥数页说明某个主题，有些技巧可能稍长。相关的技巧组织在一起形成章，但阅读它们的顺序可以由你来定。若想了解全景图，那就一页页从头读到尾。但可随意来回翻阅——当技巧需要彼此引用时，会在文中明确指出。

一开始的讨论跟代码密切关联：第 1 章“编程生产”从你擅长的编程出发，就如何让代码随时可以用于生产环境提供了指导。没有人想让充满 Bug 的代码面市，但在行业规模的项目上确保代码正确并经过良好测试却是不小的挑战。

紧随而至的第 2 章“整理工具”将帮你改善工作流程。你需要跟他人协调，自动化构建，并在工作过程中学习新技术。此外，你还要输入“成吨”的代码。值得提前对工具有所投入。

随后，我们会进入事情更麻烦的一面。陪伴你度过此生的那位经理就是你自己，第 3 章“自我管理”让你开始注意诸如压力管理和工作绩效等这类问题。

没有程序员是孤立的，第 4 章“团队协作”关注与他人的合作。不要低估人员技能——没错，擅长使用计算机是雇佣你的原因，但编程行业是一项团体活动。

接着，我们将了解宏观景象。第 5 章“走进公司”考虑了典型高科技公司的所有组成部分，以及你在整个公司中的位置。它最终试图回答，“这些家伙成天都在忙什么？”

软件企业充满风险。第 6 章“留意你的企业”谈论谁以及为何要支付你薪水，软件项目的生命周期，以及你的日常编程工作如何随那个生命周期发生改变。

最后，第 7 章“改善”将放眼未来。日语“改善”（kaizen）是一种持续改进的哲学，在我们分手之前，我希望看到你走在那条道路上。

本书约定

我经常在包含示例代码的技巧里使用 Ruby 编程语言，我选择 Ruby 仅仅是因为它简洁易读。若不懂 Ruby 也别担心，代码的意图应该一目了然。这些例子意在阐述适用于任何编程语言里的高层原则。

贯穿全书，你会看到题为“行业观点”的栏目。这些都是来自行业专家的声音，这些人是早于你走上这条道路的程序员和经理。每位贡献者都有超过 10 年的经验，因此请慎重考虑他们的建议。

从白带到黑带（再回到白带）

在整本书里，我会在你需要应用某条技巧的时候使用武术带来表示。带子颜色背后的故事要比武术本身更有意义。学生入门从白带开始，代表纯洁。同理，白带技巧适用于非常初级的阶段。

经过几年修行，带子变或褐色了。褐带代表中间阶段，坦白地讲，这时的带子是有点脏。就本书而言，我期望褐带主题对于工作第 2 ～ 5 年之间的人有帮助。

随着武术家修行日深，带子颜色越来越深，直到成为黑色。这时的武术家被冠以大师的称号。就本书而言，我设置的时间点相对较早，黑带主题在入行至少 5 年时派上用场。在实际生活里，真正的大师级水准更多地开始于第 10 个年头。

要是新大师继续使用黑带会怎样？它会磨损，被日光漂白……重新变回白色。对于专业知识，过去的大师们早就发现了一些心理学家只在最近才研究的经验之谈：只有达到某一水准，你才会知道你不知道的东西。接下来，你又一次开始新的修行之旅。

联机资源

下面是与本书相关的网页地址：

http://pragprog.com/titles/jcdeg

你可以在这里跟我和其他读者一起参与论坛讨论，查看勘误表，以及汇报你发现的任何错误。

前进

关于本书，已经唠叨得够多了。你坐在工作台旁心怀疑问“下面该干什么？”，而你的老板正奇怪你为何还未开工。那么，让我们上路吧！

致谢

首先，我必须感谢我那永远充满耐心的编辑：Susannah Davidson Pfalzer。若非她的英明指导、勉励，以及偶尔跑到我背后捣乱让我继续前进，本书可能就要泡汤了，衷心地感谢她帮助我这个第一次写书的人把这本书带到这个世界上。

接下来，还要感谢无数位提供了巨大帮助的审阅者，他们中既有刚入行的程序员新人，也不乏行业老兵。他们读过（或者应该说忍受）本书的早期草稿并给出了自己的观点、经验和修正。我要感谢 Daniel Bretoi、Bob Cochran、Russell Champoux、Javier Collado、Geoff Drake、Chad Dumler-Montplaisir、Kevin Gisi、Brian Hogan、Andy Keffalas、Steve Klabnik、

Robert C. Martin、Rajesh Pillai、Antonio Gomes Rodrigues、Sam Rose、Brian Schau、Julian Schrittwieser、Tibor Simic、Jen Spinney、Stefan Turalski、Juho Vepsäläinen、Nick Watts 和 Chris Wright。他们用勤奋和彻底的评审大大提升了本书的质量。

从一开始，几位朋友和合著者就纵容我紧追不舍、一遍又一遍地征询他们的意见，他们包括 Jeb Bolding、Mark“The Red”Harlan、Scott Knaster、David Olson、Rich Rector 和 Zz Zimmerman。我真的很感激他们的容忍。

最后，额外的感谢要献给我的两个最大粉丝。我的女儿 Genevieve，恩赐了我无法陪她而用来写书的许许多多夜晚。还有我的妻子 Daria，不仅让我有时间写作，而且第一时间购买并阅读了本书的 beta 版本——从晚上 10 点开始一气读完，有点出人意料。本书是我坐在晚餐桌旁一直考虑的点子，因此她也贡献了自己的想法和观点。同时，在整个过程中，她也提供了支持和鼓励。

Daria 和 Genevieve，没有她们，我根本没法完成本书，衷心地感谢她们。

目　录

第三部分　公司的世界

第四部分　放眼未来

第一部分

专业编程

第 ① 章

编 程 生 产

将编程作为一种娱乐消遣，你会轻易忽略像边界情况处理、错误报告等这类事情，它们确实有点麻烦。可一旦你从事编程生产（更别提要养家糊口）你就不能走捷径。

编写生产质量级别的代码似乎是一个明摆着的目标，但计算机行业却费了不少时日才弄明白正确的实现之道。例如，Windows 95 曾经有个 Bug 会让操作系统在连续运行 49.7 天之后挂起——但是该 Bug 花了 4 年时间才暴露，有 Bug 这件事本身并不特别让人觉得惊讶，时间之所以这么长是因为其他 Bug 在不到 49.7 天的时候就让 Windows 95 崩溃了。⊖

通往高质量代码的道路有两条，你可以二选一：一开始就内置质量，或者事后再敲打它。前者需要你在日复一日的编码中遵循众多戒律；后者则要求大量测试，到头来，在自以为完工之后，你会发现还有很多工作要做。

事后敲打（beat-it-in-afterward）是常见的工作方式，行业占统治地位的瀑布开发方法就是这样：规格说明、设计、构建、测试。测试是最后的步骤。产品来到测试部门，很快就崩溃了。于是，又回到工程部门，修复 Bug。接着，把另一版提交给测试部门，又由于其他原因崩溃。就这样，来来回回，许多月（甚至是数年）流逝。

本章大部分内容都聚焦于内置质量的技术，因为它是一种可以让你对自己的产品有信心、给产品添加新特性，并长年维护产品的构建方法。当然，构置生产质量级别的软件并不是一本书就可以完全覆盖的主题，而且它的范畴也要比测试大得多。不管怎样，本章仅限于讨论对改进代码质量可以起到立竿见影效果的那些事情：

- 深入具体实践之前，我们会从“技巧 1：敲打代码”开始，帮你建立正确的思维方式。

⊖ http://support.microsoft.com/kb/216641

- 接下来是“技巧 2：坚持正确”，我们将关注验证代码正确性的方法。
- 你还可以用另一种方法；在“技巧 3：测试驱动设计”中，我们尝试从测试出发，使用这些测试来驱动设计。
- 很快，你就会被巨大的代码库给弄得晕头转向。“技巧 4：驯服复杂度”尤其适合外表吓人的生产规模的软件项目。
- “技巧 5：优雅地失败”会把我们带离愉悦之路，在那里，你的代码需要应对意想不到的问题。
- 在事情才真正变得棘手的时候，我们会小憩一会儿：“技巧 6：确定风格”帮助你让代码保持美观，从长远来看，它对你的帮助超乎想象。
- 回到困难的部分。“技巧 7：改善遗留代码”处理你从前辈那里继承而来的代码。
- 最后，在“技巧 8：代码评审要早且多”中，你将和你的团队一起来保证你的代码随时可供部署。

关于这里没有谈及的内容

限于篇幅，还有其他一些对编程生产有促进的内容我并没有提及，而且在很多行业内都有你需要满足的领域相关的标准包括如下一些示例：

- 预防恶意代码、网络活动和其他安全性问题的防御性编程。
- 保护用户的数据不会因硬件和系统故障、软件 Bug 和安全破坏而受损。
- 部署并在面临巨大负载时软件性能的向外扩展。

……

向资深程序员求教：除了写出能工作的代码外——要一贯保持——还要做什么才能让你的代码合格？

技巧 1

敲打代码

[白带] 只要编写生产代码，你就要证明它经得起推敲。

你可能认为编写可靠代码是再明显不过的工作要求了。招工广告上不可能写：“急聘：具备良好工作态度、团队合作精神和桌上足球技巧的程序员。有则更佳：会编写可靠的代码。”可有问题的程序还是有这么多，怎么回事？

在深入探讨保证代码质量的日常实践之前，让我们先讨论“编写可靠代码”的含义。它不仅仅是一份实践清单，它还是一种思维方式。在把产品交到客户手中之前，你必须敲打自己的代码和整个产品。

客户终究敲打你的产品，以一种你不曾预料到的方式使用它。他们用它的时间会很长，而且会在你没有测试过的环境里用它。你必须考虑的问题是：打算让客户发现多少 Bug ？

你现在对代码敲打的次数越多，在交到客户手中之前，能清除掉的 Bug 就越多，留给客户的 Bug 就越少。

质量保证的形式

尽管本章大部分内容都关注于代码级的质量和单元测试，但品质保证却是一个要大得多的主题。让我们考虑一下产品需要经受的考验。

代码评审

保证代码质量最简单的方法就是让另一个程序员去读它。别出心裁的评审过程并没有必要，而且就连结对编程也算是一种形式的实时代码评审。团队将利用代码评审捕获 Bug，贯彻编程风格和标准，同时在团队成员间传播知识。我们将在“技巧 8：代码评审要早且多”中讨论代码评审。

单元测试

在你一个类接着一个类、一个方法接着一个方法地构建应用的业务逻辑时，验证代码的最佳方式就是单元测试。这种内部零件级的测试被设计用来对逻辑的各部分单独验证。我们将在“技巧 2：坚持正确”和“技巧 3：测试驱动设计”中讨论它们。

接受测试

单元测试立足于由内而外地审视产品，接受测试则被设计成模拟真实世界的用户，代表他们与系统交互。理想状况下，它们是自动执行的，而且以某种叙述式的风格书写出来。例如，某银行自动柜员机应用会有类似这样的接受故事：若我的活期存款为 0，当我在 ATM 的“活期存款”中选择“取款”时，那么我应该看到“对不起，今天的晚餐吃泡面吧。”

它不像莎翁著作那样文采飞扬，但这些测试操练了整个系统：从用户界面一直到业务逻辑。无论它们是自动执行的，还是人工执行的，你的公司需要知道——在任何客户使用它之前——所有系统组件正在像预期的那样协调工作。

负载测试

负载测试将产品置于真实的压力条件下，然后度量它的响应。例如，某网站可能需要在数据库有 100 万条记录的条件下在 100 毫秒内展示指定页面。这些测试将揭示正确但不恰当的行为，如需要线性伸缩但却以指数级别伸缩的代码。⊖

定向探索测试

接受测试覆盖了产品的所有指定行为，它可能来自于产品需求文档或会议。但程序员通常还是有办法使之崩溃——总有些黑暗角落被规格说明疏忽掉。定向探索测试就是要将这些边界情况挖出来。

⊖　例如，期望响应时间与负载之间的关系是线性关系，但实际却是指数关系。——译者注

“全系统”测试的范围究竟有多大？

我曾耗费数年编写工业机器人的控制软件。由于单元测试会模拟电动机的运动，我得以能够在工作站上测试业务逻辑。全系统测试当然需要运行在真正的机器人上。

就机器人来讲，好的事情是你能看到自己的代码能工作，不太好的事情是你能看到（和听到，有时甚至闻到）代码的失败。但更重要的是，机器人不是一个完美的环境。每个机器人都不同——它是上千个机电零件的组合体，每个零件多少都有些差异。因而，在多个机器人上测试非常关键。

这条经验对于更传统的系统同样适用：供应商的软件可能崩溃，网络会有延迟，硬盘会取出坏数据。公司的测试实验室应该模拟这些非理想环境，因为产品最终将会在客户手中遇到这些状况。

这种测试通常是人工执行的，可能是程序员自己，用于探索和发现问题。但最初探索之后，任何有用的测试就会被加到接受测试套件之中。

该测试有一个专业化的变种，如安全审计。在这些情况下，专业测试人员会利用他们的领域知识（可能也包括代码评审）来指导他们的测试。

机构测试

硬件产品需要不同的机构认证：FCC 度量电磁辐射，确保产品不会导致无线电干扰；美国保险商实验室（UL）检查当你将产品置于火上或舔电池电极时会发生什么。这些测试都在新产品发布之前进行，每次硬件变化都会影响认证。

环境测试

硬件产品的运行温度和湿度也需要在推至极限时测试。这些测试是用环境室来完成的，它可以同时控制这两个因素；当产品在其间运行时，它会经历所有四种极限条件。

白盒和黑盒

你会听到白盒测试和黑盒测试这两个术语。在白盒测试中，你能查看程序的内部结构，判断是否工作正常。单元测试就是这样的好例子。

黑盒测试则相反，它从客户角度审视产品，不关心内部状况，只关心产品的外部行为是否正确。

兼容性测试

一旦产品需要跟其他产品进行互操作（如某字处理程序需要跟其他字处理程序交换文档），这些兼容性的论断就需要定期验证。它们可能会访问一组已保存的文档，也可能会实时地将你的产品连接到其他产品上。

耐久性测试

你会注意到这里提到的大多数测试都是尽量频繁且快速地运行。可有些 Bug 只会在一段时间的使用之后现身。前面提到的 49.7 天的 Bug 很好说明了这一点——它源于每毫秒递增的 32 位计数器，在 49.7 天之后，它会从最大值反转成 0。[⊖]测试若不持续运行上一会儿，你就无法发现类似 Bug。

Beta 测试

产品在这一阶段被送到了真实客户手中——他们知道自己要参加测试，并同意发现问题时提交报告。Beta 测试的目的就在于我们在本技巧一开始讨论的：Beta 测试者将以你意想不到的方式使用产品，试用它一段时间，并在你没有测试过的环境中测试产品。

运行中测试

公司可能会在产品上市之后继续测试。尤其是硬件产品，如偶尔从制造线上拔掉一个单元并证明制造线能工作正常是一种很有用的方法。这些运行中测试的设计目的就是为了捕获因零件或装配过程中的变化而导致的问题。

⊖　以 Windows 上的 GetTickCount() 为例，假设为无符号计数器，2^{32} = 4 294 967 296 毫秒 = 49.7 天。

实践 VS 思维方式

你的团队可能采用类似“所有代码都必须有单元测试”或“所有代码必须先评审后检入（check in）”的实践。但这些实践没有一个能保证代码坚若磐石。想想若公司根本就没有采用一个质量实践，这种状况下该怎么做，即你将如何敲打代码以保证它的可靠性？

这是在继续深入之前你需要建立的思维方式。提交可靠的代码。质量实践只是达到目的的一种手段——最终的裁判是客户手中产品的可靠性。你想让你的名字跟市面上满是 Bug 的垃圾产品挂钩吗？不，当然不想。

行动指南

- 在上述所有形式的测试中，你的公司采用了哪些？在源代码中寻找单元测试，向测试部门询问接受测试计划，问问 Beta 测试是如何进行的以及向哪个部门提交反馈。再问下资深工程师：这是否足以保证客户有一个平滑的体验？
- 在定向探索测试上多花些时间，哪怕你的“方向”有点儿模糊。实际用一下产品，看看你是否可以让它崩溃。如果可以，那就相应地记下 Bug 报告。

技巧 2

坚持正确

[白带] 从编码第一天起就必须牢记这些。

分辩玩具程序的对错并不难。factorial(n) 有没有返回正确的数字？检查很简单：一个数字进去，另一个数字出来。但大型程序里潜在很多输入——不光是函数参数，还有系统内部的状态——以及许多输出或其他副作用。检查就没那么轻松了。

隔离和副作用

教科书喜欢用数学问题作为编程示例，部分是因为计算机擅长处理数学问题，但更多是出于单独论证数字很容易的缘故。调用 factorial(5) 一整天，返回结果总是相同的。可网络连接、磁盘文件或者（尤其是）用户都有些不那么好预测的棘手行为。

若函数会改变除局部变量以外的事物（如向文件或网络套接字写数据），它就被称为有副作用。相反，纯（pure）函数总是在给定相同输入参数的情况下返回相同值，不改变任何外部状态。纯函数显然比有副作用的函数容易测试得多。

大多数程序都是纯和不纯代码的混合体，但认真区分两者的程序员并不太多。你可能曾见过类似这样的代码：

ReadStudentGrades.rb

```
def self.import_csv(filename)
  File.open(filename) do |file|
    file.each_line do |line|
      name, grade = line.split(',')

# 将数字成绩转变成字母成绩
grade = case grade.to_i
  when 90..100 then 'A'
  when 80..89 then 'B'
  when 70..79 then 'C'
  when 60..69 then 'D'
  else 'F'
end
```

```
        Student.add_to_database(name, grade)
      end
    end
  end
```

这个函数做了 3 件事情：逐行读入一个文件（不纯），做些分析（纯），然后更新全局数据结构（不纯）。这种写法让你无法方便地测试任何一部分。

话说到这份上，显然应该隔离每项任务，好让它们可以单独测试。我们将在后面的“交互”一节简短讨论文件那部分代码。现在让我们先把分析部分的代码抽成单独的方法。

ReadStudentGrades2.rb

```
def self.numeric_to_letter_grade(numeric)
  case numeric
    when 90..100 then 'A'
    when 80..89 then 'B'
    when 70..79 then 'C'
    when 60..69 then 'D'
    when 0..59 then 'F'
    else raise ArgumentError.new(
      "#{numeric} is not a valid grade")
  end
end
```

现在，numeric_to_letter_grade() 是一个纯函数了，可以方便地独立测试：

ReadStudentGrades2.rb

```
def test_convert_numeric_to_letter_grade
  assert_equal 'A',
    Student.numeric_to_letter_grade(100)
  assert_equal 'B',
    Student.numeric_to_letter_grade(85)
  assert_equal 'F',
    Student.numeric_to_letter_grade(50)
  assert_equal 'F',
    Student.numeric_to_letter_grade(0)
end
def test_raise_on_invalid_input
  assert_raise(ArgumentError) do
    Student.numeric_to_letter_grade(-1)
  end

  assert_raise(ArgumentError) do
    Student.numeric_to_letter_grade("foo")
  end

  assert_raise(ArgumentError) do
    Student.numeric_to_letter_grade(nil)
  end
end
```

这个例子可能有点简单，但若业务逻辑复杂且又深埋在一个具有 5 个不同副作用的函数里，情况又如何呢？（答案：没法很好地测试它。）区分开纯和不纯代码有助于测试正确性，这种方法既适用于新代码，也适用于维护遗留代码。

交互

现在该如何处置那些副作用？如果代码充斥类似这样的结构："若在测试模式，就不要真正连接数据库……"将会让人痛苦不堪。换一种做法，大多数语言都有一种创建测试替身（test double）的机制，它可以代替函数需要使用的资源。

让我们重写上面的例子，这样一来，import_csv() 就只负责文件处理，把其余工作交给 Student.new()：

ReadStudentGrades3.rb

```
def self.import_csv(filename)
  file = File.open(filename) do |file|
    file.each_line do |line|
      name, grade = line.split(',')

      Student.new(name, grade.to_i)
    end
  end
end
```

我们需要的是一个文件的测试替身，它将拦截对 File.open() 的调用，并产生一些模拟数据。我们需要对 Student.new() 采用同样的做法，理想情况下会在拦截调用时验证传入它的数据。

Ruby 的 Mocha 框架完全可以让我们做到这一点：

ReadStudentGrades3.rb

```
def test_import_from_csv
  File.expects(:open).yields('Alice,99')
  Student.expects(:new).with('Alice', 99)

  Student.import_csv(nil)
end
```

这个例子阐述了测试方法间交互的两点内容：

- 单元测试不应该污染系统状态，留下大量的无效文件句柄、数据库中的对象或其他垃

圾对象。测试替身框架可以让你避免这一点。

- 这类测试替身称为模拟对象（mock object），它会验证你给它设定的期望值。若 Student.new() 未被调用或调用参数不同于测试中的指定值，Mocha 将让测试失败。

当然，Ruby 和 Mocha 让问题极度简化了。对于我们这些受困于上百万行 C 程序的弟兄们该怎么办？就算是 C，它也能使用测试替身，只不过比较麻烦。

问题可以概括为：如何在运行时用一组函数替换另一组？（要是你认为“这听起来像是动态分派表”，那你说对了。）继续使用打开和读取文件的例子，这里给出了一种方法：

TestDoubles.c

```
struct fileops {
    FILE* (*fopen)
        (const char *path,
         const char *mode);


    size_t (*fread)
        (void       *ptr,
         size_t      size,
         size_t      nitems,
         FILE       *stream);
    // ...
};

FILE*
stub_fopen(const char *path, const char *mode)
{
    // 仅返回伪文件指针
    return (FILE*) 0x12345678;
}

// ...

struct fileops real_fileops = {
    .fopen = fopen
};

struct fileops stub_fileops = {
    .fopen = stub_fopen
};
```

fileops 结构有一个指向标准 C 库 API 的函数。在 real_fileops 结构中，我们用真实函数填充这些指针。在 stub_fileops 里，它们指向我们自己的替身版本。使用这个结构跟调用一个函数没有太大的区别。

TestDoubles.c

```
// 假设 ops 是一个函数参数或全局变量
struct fileops *ops;
ops = &stub_fileops;

FILE* file = (*ops->fopen)("foo", "r");
// ...
```

现在，只需重新给指针赋值，程序就能在“真实模式”和“测试模式”间来回切换了。

类型系统

如果在代码中引用类似 42 的值，它到底是数字、字符串，还是其他事物？假若有一个类似 factorial(n) 的函数，应该给它传入什么类型的参数，输出值又应该是什么？元素、函数和表达式的类型非常重要。语言对类型的处理方式称为它的类型系统（type system）。

类型系统是书写正确程序的重要工具。例如，在 Java 中，你可以写出像这样的方法：

```
public long factorial(long n) {
  // ...
}
```

在这个例子里，读者（你）和编译器都可以轻易地推断 factorial() 应该接受一个数字，返回一个数字。Java 是静态类型的，因为它在代码编译时检查类型。试图传入一个字符串将导致编译失败。

价值 6 千万美元的 Break 语句

1990 年 2 月 15 日，AT&T 的电话网络照常在忙碌地运转。直到下午的 2:25。当时，一个电话交换机执行了一个自检操作，然后自行复位。交换机并不经常复位，但网络可以处理这种情况，而且交换机只要 4 秒钟就能完成复位，恢复正常运行。只是这次有点不同，其他交换机也开始复位了，短短几秒内，AT&T 的全部 114 台主干交换机都在不停地自行复位。强大的 AT&T 电话系统顿时停止了工作。

调查发现，当第一台交换机复位自己时，它告诉邻近的交换机它正在恢复正常运转。消息交换导致了邻近交换机的崩溃。它们依次自动复位，向它们的邻近交换

机发送恢复运转的消息，以此类推……由此导致了无尽的复位 / 恢复 / 复位循环。

AT&T 的工程师花了 9 个小时让电话系统恢复正常。据估计，服务中断导致的通话中断让 AT&T 损失 6 千万美元，对于那些依赖电话完成业务的人们，造成的经济损失更是无法估量。[⊖]

是什么导致了这个问题？是一条错误的 break 语句。有人用 C 写出了下面的语句：

```
if (condition) {
    // do stuff...
}
else {
    break;
}
```

代码的表面意思是："若条件为真，那么就工作；否则，什么也不做。"但在 C 语言中，break 语句并不会让 if() 语句中断；它中断的是像 while() 或 switch() 这样的语句块。实际发生的事情是：break 过早退出了包含块，破坏了数据结构，导致电话交换机复位。由于所有电话交换机都运行相同软件，且该 Bug 存在于用来处理来自对等点的复位恢复消息的代码中，这个错误在整个网络中不断地扩散开来。

将它跟 Ruby 比较：

```
def factorial(n)
  # ...
end
```

这个方法可接受的输入是什么？光看函数签名，你无法判断。Ruby 是动态类型的，因为它直到运行时才验证类型。这让你拥有了巨大的灵活性，而且意味着某些会在编译时捕获的错误将直到运行时才能被捕获。

两种方式都有自己的优缺点，但要获得正确性，请记住以下几点：

- 静态类型有助于传播函数的正确使用，避免了其被滥用的危险。若阶乘函数接受一个长整型，并且返回一个长整型，那么编译器不会允许你传入一个字符串作为替代。

⊖ http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html

但它不是魔力弹：假如调用 factorial(-1)，类型系统不会报错，于是错误将在运行时发生。

- 要想好好利用静态类型，你就必须按它的规则行事。常见的例子是在 C++ 里使用 const：当你开始使用 const 来声明某些事物不能改变时，编译器就真的会对每个正确声明其参数常数性的函数变得挑剔起来。若你能完完全全遵循这些规则，它就变得有价值；但只要你的承诺低于 100%，它就会变得歇斯底里。
- 动态类型语言可能会让你行动迅速，并且放松对类型的要求，但要是用字符串调用 factorial() 还是没什么意义。你需要使用面向契约的单元测试，这将在“技巧 3：测试驱动设计”中讨论，确保你的函数能正确检查其参数的合理性。

不论语言用的是哪种类型的系统，都要养成记录你对每个参数的期望的习惯——它们通常无法像 factorial(n) 例子那样具有自解释性。参见“技巧 6：确定风格”，进一步了解对文档和代码注释的讨论。

100% 覆盖率是一派胡言

回答“我是否充分进行了测试？”这一问题的常用（但有缺陷）指标是代码覆盖率。即在运行单元测试时执行了百分之几的应用代码？理想情况下，运行单元测试时，应用的每行代码至少运行一次，即覆盖率是 100%。

低于 100% 的覆盖率意味着有代码未被测试到。初级程序员会假设逆命题为真：若代码达到 100% 覆盖率，它们就经历了足够的测试。可惜事与愿违：100% 覆盖率绝不意味着覆盖了所有情况。

考虑以下 C 代码：

BadStringReverse.c

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reverse(char *str) // 糟糕透顶
{
```

```
`
        int len = strlen(str);
        char *copy = malloc(len);

        for (int i = 0; i < len; i++) {
                copy[i] = str[len - i - 1];
        }
        copy[len] = 0;

        strcpy(str, copy);
}

int main()
{
        char str[] = "fubar";
        reverse(str);
        assert(strcmp(str, "rabuf") == 0);
        printf("Ta-da, it works!\n"); // 不完全
}
```

这个测试覆盖了 reverse 函数的每行代码。这是否表明函数就正确了呢？非也：malloc() 分配的内存从来就没被释放，而且被分配的缓存只有 1 个字节，太小了。

不要满足于 100% 的覆盖率：它跟代码或测试的质量风马牛不相及。书写优秀的测试就像书写优秀的应用代码，需要思考、勤勉和优秀的判断。

低于 100% 覆盖率

有些情况极其难以进行单元测试。这里有些例子：

- 跟硬件打交道的内核驱动程序依赖代码控制之外的硬件状态改变，几乎不可能创建高保真的测试替身。
- 多线程代码可能会有时间问题，遇上它们算你走运。
- 第三方提供的二进制代码经常无法按意愿触发而返回失败。

那么，怎样从测试中得到 100% 的覆盖率？只要法术高强，完全可能做到，但这是否值得？从价值判断，其结果可能是不值得。遇到这种情况时，与团队的技术领导讨论这个问题。他们可能会想出一种不那么痛苦的测试方法。若没有其他办法，你就需要他们评审你的代码。

不要因为无法取得 100% 的覆盖率而放弃追求进步，不要将它作为回避完全测试的借

口。用测试证明代码的合理性，其他的事情则服从资深程序员给出的意见。

进阶阅读

Kent Beck 的《Test-Driven Development: By Example》（测试驱动开发）[Bec02] 仍是单元测试的基础读物。虽然它用 Java 来举例，但其原则适用于任何编程语言。（在阅读它时，试着自行解决示例问题，你可能会提出更优雅的解决办法。）我们将在“技巧 3：测试驱动设计”中讨论测试驱动的内容。

要全方位了解用 Ruby 写单元测试的技巧，Ruby 程序员应该选择《The RSpec Book》[CADH09]。

C 语言程序员应该读读《Test Driven Development for Embedded C》（测试驱动的嵌入式 C 语言开发）[Gre10] ㊀，可以了解 TDD 和构建测试用具（test harness）的技术。

围绕测试替身存在着一个命名学。像 mock 和 stub 这样的术语有特定的含义。Martin Fowler 有篇在线文章㊁很好地解释了其中的细节。

关于类型系统和使用它们构建正确代码有一整套理论，参见 Pierce 的《Types and Programming Languages》（类型和程序设计语言）[Pie02] 了解骇人的细节。此外，Kim Bruce 的《Foundations of Object-Oriented Languages: Types and Semantics》[Bru02] 特别强调了 OOP。

行动指南

- 对于你使用的每一种编程语言，都去找对应的可用单元测试框架。大多数语言都同时包含常用基础工具（断言（assert）、测试初始化（test setup）和测试后处理（test teardown））和某种支持伪对象（mock 和 stub）的工具。安装任何你需要让这些工具

㊀ 本书的中文版是《测试驱动的嵌入式 C 语言开发》，机械工业出版社出版，ISBN:978-7-111-36623-2。——译者注

㊁ http://martinfowler.com/articles/mocksArentStubs.html

运行起来的工具。

- 本技巧包含了一个程序的零散代码，它要完成：逐行从文件读入逗号分隔的数据，将它们分拆开，然后用它们创建对象。选择一种语言编写完成该程序，一并完成保证应用代码每行正确性的单元测试。

技巧 3

测试驱动设计

[褐带] 你可能不会从一开始就设计新代码，但机会很快就会有了。

在技巧 2 中，我们专注于确保让代码做该做的事；在本技巧里，我们关心这样一个元问题（meta-question）：㊀“这个代码应该做什么？”

“程序员会在不知道代码该干什么之前（离知道还早得很）就开始写代码。”乍一看，这让人很费解。可一直以来我们都是这样工作的。遇到问题时，我们会停止编码，找出前进的方向。编程是一项创造性的活动，不是机械性的，如果不明确目标，就类似于画家在一块空白画布上四处涂抹，完全不知道最终作品会是什么样子。（这会不会就是有如此多代码类似于 Jackson Pollock ㊁ 的作品的原因呢？）

可编程需要严谨，测试为你提供了同时用于设计和满足严谨性的工具。

测试驱动的设计

多亏了“交互”一节讨论过的测试替身框架，你才能够从宏观编程问题出发，从任何合理的角度开始向它发起攻击。你的程序可能需要抓取一个包含客户统计数据的 XML，对其遍历，产生数据的汇总统计。你无法马上确定如何解析这个 XML，可你完全知道客户平均年龄的计算方法。没问题，模拟出 XML 解析，测试这个计算方法：

㊀ 关于另一个问题的问题，在这里的问题是针对“代码做该做的事情”而言。——译者注

㊁ Jackson Pollock（杰克逊 • 波洛克），美国抽象表现主义画家。波洛克的创作过程与众不同，他先把画布钉在地板上或墙上，然后随意在画布上泼洒颜料，任其在画布上滴流，创造出纵横交错的抽象线条效果。波洛克有时还用石块、沙子、铁钉和碎玻璃掺和颜料在画布上摩擦。他摒弃了画家常用的绘画工具，绘画时完全摆脱受制于手腕、肘和肩的传统模式，行动即兴、随意，这种方法被称为行动绘画或抽象表现主义。（摘自百度百科。）——译者注

AverageCustomerAge.rb

```
class TestCustomerStats < Test::Unit::TestCase
  def test_mean_age
    data =
      [{:name => 'A', :age => 33},
       {:name => 'B', :age => 25}]
    CustomerStats.expects(:parse_xml).returns(data)
    File.expects(:read).returns(nil)

    stats = CustomerStats.load
    assert_equal 29, stats.mean_age
  end
end
```

现在，你能写出如下代码：

AverageCustomerAge.rb

```
class CustomerStats
  def initialize
    @customers = []
  end

  def self.load
    xml = File.read('customer_database.xml')
    stats = CustomerStats.new
    stats.append parse_xml(xml)
    stats
  end

  def append(data)
    @customers += data
  end

  def mean_age
    sum = @customers.inject(0) { |s, c| s += c[:age] }
    sum / @customers.length
  end
end
```

确信计算部分已经完工之后，你就可以转到解析 XML 了。从巨大的客户数据库里选 2 个条目，只需确保格式正确即可：

data/customers.xml

```
<customers>

  <customer>
    <name>Alice</name>
    <age>33</age>
  </customer>
```

```
  <customer>
    <name>Bob</name>
    <age>25</age>
  </customer>
</customers>
```

接下来是验证解析的简单测试：

AverageCustomerAge.rb

```
def test_parse_xml
  stats = CustomerStats.parse_xml(
    canned_data_from 'customers.xml')
  assert_equal 2, stats.length
  assert_equal 'Alice', stats.first[:name]
end
```

由此你可以开始提取 XML 内容：

AverageCustomerAge.rb

```
def self.parse_xml(xml)
  entries = []
  doc = REXML::Document.new(xml)

  doc.elements.each('//customer') do |customer|
    entries.push({
      :name => customer.elements['name'].text,
      :age => customer.elements['age'].text.to_i })
  end

  entries
end
```

你可以自己灵活把握从哪里开始设计：自顶向下、自下而上或从它们之间任何一处开始。既可以从风险最大（即你最担心的）部分开始，也可以从最有把握的地方出发。

测试在这里起到了几个作用：首先，它让你可以快速前进，因为你模拟了代码与外部组件的交互。“我知道需要从 XML 中得到这个数据，但让我们假设已有其他方法完成了这项工作。”其次，测试自然地驱动了模块化风格构造——它让这种工作方式做起来更简单了。最后，测试在一旁确保你（或将来的维护者）不会无意间破坏某些功能。

测试即规格说明

在某一时刻，你对每个函数的职责有了更好的理解。这时就可以进一步明确：函数在正

常路径上应该确切完成什么工作？它不该完成什么？它应该如何失败？将测试视为一种规格说明：告诉计算机（和 5 年后需要维护你代码的程序员）你的确切期望。

让我们从一个简单的例子开始，一个阶乘函数。首先问：它应该完成什么？根据定义，n 的阶乘是所有小于等于 n 的正整数乘积。0 的阶乘是特例，其值为 1。这些规则足以用 Ruby 单元测试表示：

Factorial.rb

```
def test_valid_input
  assert_equal 1, 0.factorial
  assert_equal 1, 1.factorial
  assert_equal 2, 2.factorial
  assert_equal 6, 3.factorial
end
```

在选择测试值时，我测试了有效边界条件（0）和足以用来建立阶乘模式的值。出于说明的目的，你可以测试更多值，但这不是严格必需的。

下一个问题是：无效输入有哪些？首先想到的是负数，然后是浮点数。（严格讲，非整数和复数都有阶乘[⊖]，但让我们不要把问题复杂化。）让我们把这些约束也写出来：

Factorial.rb

```
def test_raises_on_negative_input
  assert_raise(ArgumentError) { -1.factorial }
end

def test_factorial_does_not_work_on_floats
  assert_raise(NoMethodError) { 1.0.factorial }
end
```

对于负整数，我选择抛出 ArgumentError 异常；对于在任何其他类型上调用 factorial，我选择让 Ruby 抛出 NoMethodError。

这是一个合理、完整的规格说明。事实上，代码本身在这里已经很大程度上把自己给写出来了。（动手吧，写一个可以通过测试的阶乘函数。）

⊖ http://en.wikipedia.org/wiki/Factorial

过度测试

程序员在开始单元测试时经常会问：我需要测试的全部值是什么？例如，对于阶乘函数，你可以测试上百个值。但这样做有没有更多的意义？没有。

因此，只需测试那些对于确定函数行为必需的值就够了。这既包含正常路径也包含错误情况。然后就收工。

除了浪费时间外，进行多余的测试会有害处吗？有：

- 单元测试的价值等同于规格说明，多余枝节难以让读者从一堆废话中分辨出规格说明的重点。
- 每行代码潜在都包含 Bug——即使是测试代码。调试不必要的测试代码会成倍地浪费时间。
- 万一决定改变模块的接口，要变动的测试将更多。

因此，只写出你需要验证正确性的代码。

进阶阅读

《Growing Object-Oriented Software, Guided by Tests》[FP09] ⊖广泛覆盖了采用 TDD 和模拟技术（mocking）的设计过程。

同前面一样，Ruby 程序员可从《The RSpec Book》[CADH09] 获益匪浅。

若觉得“测试即规格说明”听起来非常像归纳证明，那没错。你可以在《Algorithm Design Manual》（算法设计手册）[Ski97] 中更多地了解归纳证明。

⊖ 本书的中文版是《测试驱动的面向对象软件开发》，机械工业出版社出版。——译者注

行业观点：不同意见

许多人将大把时间花到提前设计和找出分解问题的方法——现在则变成了如何将它拆分成类——我认为你提前做的任何决定都将是错误的。

我的建议不同于传统观点：尽早开始编码。

当你开始看一个问题时，先犯错误。在编程时，我会做一个只包含了几个大类的原型。一旦对问题有了更好的理解，我就开始编写产品代码。现在很多时候，程序员会提前把问题分解成类，然后他们会把实现强行塞入他们在没有足够信息时创建的这个结构。

——Scott“Zz”Zimmerman，资深软件工程师

行动指南

从本技巧一开始，我们就用到了一些以 XML 编码的数据。这在行业里是极其常见的任务，因此练习加载和保存 XML 是非常有益的。

从非常简单的结构开始，如上述的客户列表。由于你想专注于处理解析结果，那就用现成的解析器完成实际解析，如 Ruby 的 REXML。在马力全开编写任何代码之前，思考你将为 XML 加载函数构造的测试：

- 若列表中没有客户会怎样？
- 如何处理空白字段？
- 对于无效字符该如何处理，如年龄字段中有字符？

随着这些问题的回答和书写成测试，现在可以编写加载器函数了。

附加题：为操纵客户列表并将结果存回文件构建一些测试。你可以使用像 Ruby Builder 这样的 XML 生成器。

技巧 4

驯服复杂度

[白带] 你从入职第一天起就要应对复杂代码。

若是还未遇到过无法理解的程序，那说明你编程的年头还不够长。在行业里，要不了多久你就会碰到让人发懵的混乱代码：巨兽、面条工厂、来自地狱的遗留系统。我曾接手过一个程序，它的前任在听说要增加一个分量不轻的新特性时，选择了辞职。（我并不怪他。）

软件系统的复杂度是不可避免的。有些问题就是很难，它们的解决方案很复杂。然而，你在软件中找到的大多数复杂度是我们自己造成的。在《The Mythical Man-Month》（人月神话）[Bro95] 里，Fred Brooks 将复杂度的两个来源分成必然（necessary）复杂度和偶然（accidental）复杂度。

这里有一种区分必然复杂度和偶然复杂度的思考方法：什么复杂度是问题域固有的？假设你面对的是一个日期 / 时间处理代码散落各处的程序。在处理时间时，存在一些必然复杂度：每月的天数不同，必须考虑闰年，等等。但多数我碰到的程序充斥着大量与处理时间相关的偶然复杂度：用不同格式保存的时间，加减时间的新奇（同时也是充满 Bug 的）方法，不一致的时间打印格式，说都说不完。

复杂度的死亡螺线

编程时常会遇到这种情况：产品代码库中的偶然复杂度渐渐压倒必然复杂度。情况在某一时刻会自我放大，我称这种现象为复杂度的死亡螺线，如图 1 所示。

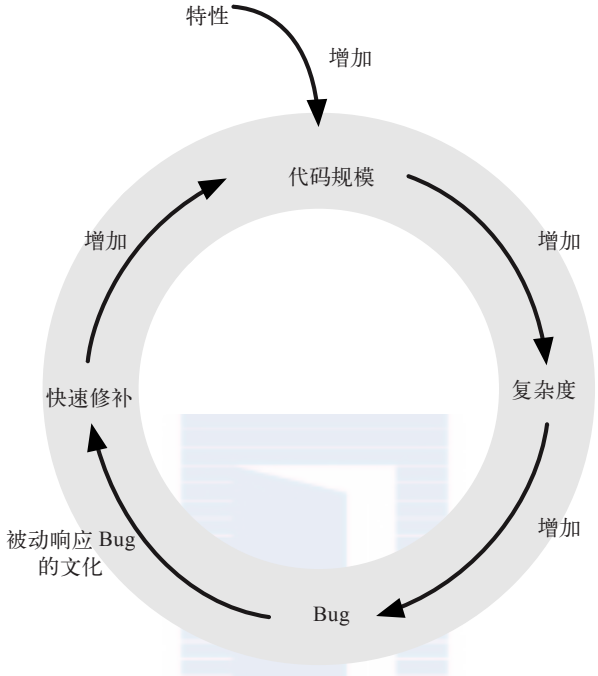


图 1　复杂度的死亡螺线

问题 1：代码规模

构建产品时，它的代码规模最终将远超任何在学校或消遣项目中所遇到的。行业中的代码库的度量结果从成千到上百万代码行（Line of Code, LOC）不等。

John Lions 在《Lions' Commentary on UNIX 6th Edition》[Lio77] ⊖一书中写道：单个程序员能够理解和维护的程序大小的实际限制规模是 1 万行代码。于 1975 年发布的 UNIX 第 6 版的规模大约是 9000 行代码（不算机器特定的设备驱动程序）。

相比而言，Windows NT 在 1993 年有 4 百万 ~5 百万行代码。10 年后，Windows Server 2003 配备了 2000 名开发人员和 2000 名测试人员，他们管理多达 5 千万行代码。⊜大多数行业项目并不像 Windows 那样巨大，但它们也都轻易地跨过了 Lions 设定的 1 万行代码的警戒

⊖　本书的中文版是《莱昂氏 UNIX 源代码分析》，机械工业出版社出版。——译者注

⊜　http://en.wikipedia.org/wiki/Source_lines_of_code

线。这样的规模意味着公司内部没有人能理解整个代码库。

问题 2：复杂度

随着代码规模的增长，最初想法的概念优雅性消失了。曾经对于车库中两个小伙水晶般清澈的想法变成了大批开发人员艰难跋涉其中的阴暗沼泽。

复杂度并不是代码规模的必然产物。大型代码库完全有可能被拆分成许多模块，其中每个模块都有清晰的用途、优雅的实现和为人熟知的与邻近模块的交互。

然而，即使设计良好的系统也会在它们变大时变得复杂。一旦没有一个人可以理解整个系统，这时多个人必须去理解系统中自己那部分——且没有人的理解跟其他人是完全一样的。

问题 3：Bug

产品复杂度飙升，Bug 也就不可避免地出现了。这是注定的——就算是伟大的程序员也不是完人。但每个 Bug 并非生而平等：高度复杂系统里的那些 Bug 尤其难觅踪迹。总是听到程序员说：“真搞不懂，伙计，系统刚刚崩溃了。”欢迎来到这糟糕的调试世界！

问题 4：快速修补

问题并不在于产品是否有 Bug——它肯定有，关键在于工程团队在出现 Bug 之后如何响应。在推出产品的压力之下，大多数程序员经常求助于快速修补。

快速修补是给问题打补丁，而非解决其根本原因。甚至常常不寻找根本原因。这里有个例子：

LOC 度量的是规模，不是进度

经理们总想度量各种事情，既然构建软件产品是写代码，那么用它的代码行来度量产品的进度也说得过去——从表面上看是如此。但这是完全错误的度量方法，因为优秀的程序员会寻找优雅的解决方案，而优雅的解决方案往往比暴力方法使用更少的代码。

代码行对于度量某件事是有益的，但不是进度：它度量的是规模。比尔•盖茨认为：用代码行度量编程进度就像用重量度量飞机的制造进度。[⊖10]

就算不是航天工程师也知道应该将飞机尽量造得轻一些——任何多余的重量都会降低飞机的效率。可飞机还是很重。一架空客A380重达61万磅，它还搭乘大约650名乘客。（相形之下，一架塞斯纳172的重量只有可怜的1620磅，运载4名乘客——但不体面，因为这种飞机中不会有饮料推车。）

同样，一个特性丰富的产品肯定会包含大量代码，这在所难免。但产品应该尽量轻装上阵，任何多余的代码行只会拖累它未来的开发。

程序员：在试图往网络队列中放入一个任务（job）且队列在10秒内无响应时，程序崩溃了。

经理：重试队列操作100次。

根本原因是什么？天知道，只要重试次数够多，你就可以掩盖任何问题。但如车身修补一样，某一位置的霸道胶水（Bondo）[⊜]比实际残留的车本身部件还要多。

更难找的问题发生在补丁并没有解决问题根本原因的时候，问题通常根本没有消失——它只是转移到别处。在前面的对话中，重试100次可能很好地掩盖了问题，但万一需要101次重试怎么办？经理只是随便捏了个数字，这种膏药式修补只会让问题更难查。

沿着“快速修补”上行，我们现在得到了一个增加代码规模的完整闭环。

走向清晰

提起复杂的反面，人们通常会想到简单。但由于领域的必然复杂度，我们并不是总能写出简单的代码。应对复杂更好的方法是清晰。你是不是明白自己的代码要做什么？

⊖ http://c2.com/cgi/wiki?LinesOfCode

⊜ Bondo，霸道胶水，美国万能胶品牌。——译者注

明确两点会有助于我们减少软件偶然复杂度：清晰思考和清晰表达。

清晰思考

在分析问题的原因时，我们试图做出像“保存时间的方式应该只有一种”这样的清晰陈述。那为何 UNIX C 代码里还混杂着像 time_t、struct timeval 和 struct timespec 这样的结构呢？[⊖]那并不是太清晰。

如何调和你的清晰陈述和 UNIX 计时功能的复杂度？你需要隔离复杂度，或将其抽象到单个模块中。在 C 里，这可能是结构体和操作它的函数；在 C++ 里，它会是一个类。模块化设计让程序的其余部分可以用一种清晰的方式推导时间，而不用了解系统计时功能的内部机制。

一旦能将时间作为程序的一个单独模块进行对待，你也就能证明你的计时机制的正确性。完成这一工作的最佳方式就是单独测试，但是同行评审或书写规格说明也行。当一组逻辑是隔离的而不是内嵌在一大段代码体内时，它的测试和严格证明要容易得多。

清晰表达

随着你清晰地思考模块并将它与其余程序隔离，最终程序也就能更清晰地表达它的用途。处理问题域的代码应该真正专注于问题域。

将辅助代码抽出放入自己的模块之后，剩余逻辑读起来应该越来越像问题域的规格说明（虽然有更多分号）。

让我们看看前后对比。我已经无数次看到过如下这种 C++ 代码：

Time.cpp

```
void do_stuff_with_progress1()
{
    struct timeval start;
    struct timeval now;

    gettimeofday(&start, 0);
```

⊖ http://en.wikipedia.org/wiki/Unix_time

```
    // 干活，每半秒打印一条进度消息
    while (true) {
        struct timeval elapsed;
        gettimeofday(&now, 0);
        timersub(&now, &start, &elapsed);

        struct timeval interval;
        interval.tv_sec = 0;
        interval.tv_usec = 500 * 1000; // 500ms

        if (timercmp(&elapsed, &interval, >)) {
            printf("still working on it...\n");
            start = now;
        }
        // 干活……
    }
}
```

循环的关键是“干活”部分，但在实际干活之前有 20 行的 POSIX 计时代码块。这并没有什么不对，但……就没有一种方法让循环保持对其问题域而不是对计时的关注吗？

让我们把所有时间代码放入它自己的类：

Time.cpp

```
class Timer
{
public:
    Timer(const time_t sec, const suseconds_t usec) {
        _interval.tv_sec = sec;
        _interval.tv_usec = usec;
        gettimeofday(&_start, 0);
    }
    bool triggered() {
        struct timeval now;
        struct timeval elapsed;

        gettimeofday(&now, 0);
        timersub(&now, &_start, &elapsed);

        return timercmp(&elapsed, &_interval, >);
    }

    void reset() {
        gettimeofday(&_start, 0);
    }

private:
    struct timeval _interval;
    struct timeval _start;
};
```

我们现在可以简化循环了：

Time.cpp

```
void do_stuff_with_progress2()
{
    Timer progress_timer(0, 500 * 1000); // 500ms


    // 干活，每半秒打印一条进度消息
    while (true) {
        if (progress_timer.triggered()) {
            printf("still working on it...\n");
            progress_timer.reset();
        }

        // 干活……
    }
}
```

计算机在上述两种情况下做的事情是相同的，但考虑第二个例子对程序可维护性带来的影响：

- Timer 类的测试和证明可独立于它在程序中的使用方式。
- “干活”循环内的计时有了有意义的语义——triggered() 和 reset()，而不是一堆获取、增加和比较函数。
- 现在对于计时的终止位置和（捏造的）循环实际起始位置都清晰了。

当工作在巨大丑陋的代码上时，依次考虑：这段代码想表达什么含义？它有没有办法说得更清楚一点？如果它是清晰表达的问题，你需要把那些碍事的代码段抽象出来，同前面展示的 Timer 类一样。若代码还是有点混乱，那可能是没有清晰思考的产品，需要在设计层面返工。

行动指南

聚焦于可被隔离和严格推导的一个编程方面，如计时。挖掘你正在从事的项目，识别出这样的代码段：若那部分逻辑被抽象到自己的模块，它能否表达得更清晰。

动手尝试更模块化的方法：选一组混乱的代码，分离必然复杂度和偶然复杂度。在这一刻不要操心细节，只看如何可以清晰地表达必要的业务逻辑，假设你有独立模块来处理支撑逻辑。

技巧 5

优雅地失败

[白带] 编写正常失败的代码跟编写正常工作的代码同等重要。

代码失败后会发生什么？失败总会到来。即便你的那部分写得非常完美，但有各种情况会导致整个系统失败：

- 流氓邮件守护进程驻留在计算机上，频繁发送来自某个国家的巨额中奖通知单，耗尽了所有的内存和交换区。malloc() 下次调用的返回值是 ETOOMUCHSPAM。
- Java 的第 134 001 项更新包占满了系统硬盘。调用 write()，系统返回 ESWITCHTODECAF。
- 你想从磁带上取下数据，但磁带机器人位于海边，海浪导致机器人把磁带弄掉了，于是驱动器返回 EROBOTDIZZY。
- 宇宙射线导致内存中某位翻转，使得内存存取返回 0x10000001 而非 0x1，在 memcpy() 返回 EMEMTRASHED 之后，你发现导致这出现的原因是传给了它一个非常坏的参数。

你可能会想“耶，是这样吗”，但所有这些情况都实际发生过。（真的，我就修过一个磁带机器人的控制器，因为它会在军舰开过时把磁带掉在地上。）你的代码不能天真地假设它身处于一个健全的世界——现实世界会逮住任何机会证明这个想法是错的。

代码如何失败就跟它如何工作一样重要。你可能无法修复失败，但代码至少应该尽量失败得优雅一点。

操作顺序

很多教科书里的程序环境纯洁无瑕，程序可一帆风顺地运行直到结束。非教科书程序则身处麻烦之地，环境是线程和资源的角力，怎么看都像彼此要拼出个胜负才肯罢休。

考虑以下例子：你正在创建一个客户名字和地址的列表，它将提供给标签打印机。代码接受一个客户 ID 和一个数据库连接，因此你需要从数据库查询需要的信息。你创建了一个链表，它的 add() 方法类似这样：

ListUpdate.rb

```
def add(customer_id) # BAD BAD BAD, see text
  begin
    @mutex.lock
    old_head = @head
    @head = Customer.new
    @head.name =
      @database.query(customer_id, :name)
    @head.address =
      @database.query(customer_id, :address)
    @head.next = old_head
  ensure
    @mutex.unlock
  end
end
```

（是的，我知道这个例子不真实。但请忍一忍。）

这段代码工作在一条愉快的路径上：新元素置于列表头部，然后填充列表，每一件事都让人愉悦。可万一其中一次数据库查询抛出异常怎么办？再看看代码。[⊖]

这段代码并没有优雅地失败。事实上，它还因为允许数据库失败附带地把用户列表给破坏掉。罪魁祸首是操作顺序：

- 列表 @head 和 @head.next 对于列表的完整性至关重要。在其他准备做好之前，不应该触及它们。
- 新对象应该在插入列表之前完全构造好。
- 在进行可能引起阻塞的操作时，不应该持有锁。（假设还有其他线程等待读取列表。）

⊖ 答案：首先，引表头部已经放置了新元素，因此列表头部将至少有一个字段为空。其次，列表的余下部分将无影无踪，因为列表头部的 next 字段只在数据库查询之后才更新。而且，还有一个坏处：列表在数据库查询（操作的完成时间不确定）期间保持锁定状态。

事务

前一节的例子只有一个关键状态位需要保持一致。若有多个需要这样做的情况呢？考虑在两个银行账户间转账的经典例子：

Transaction.rb
```
savings.deduct(100)
checking.deposit(100)
```

假如数据库刚好在钱被扣除时出问题，存入支票失败了，会出现什么状况？钱到哪里去了？你可能想通过将钱放回存款账户来解决问题：

Transaction.rb
```
savings.deduct(100)       # 快乐工作

begin
  checking.deposit(100)  # 失败：数据库崩溃！
rescue
  begin
    # 把钱放回
    savings.deposit(100) # 失败：数据库仍然没有反应
  rescue
    # 现在怎么办？
  end
end
```


很好的尝试，但若第二次 deposit() 失败，无计可施了。

这里需要用到的工具是事务。其目的就是允许几个操作，潜在地操作几个对象，要么完全执行，要么回滚。

事务（在这个人为的系统中）将会让前面的例子看上去像这样：

Transaction.rb
```
t = Transaction.new(savings, checking)
t.start

# 插入失败
checking.expects(:deposit).with(100).raises

begin
  savings.deduct(100)
  checking.deposit(100)
  t.commit
```

```
rescue
  t.rollback
end
```

你通常可以在数据库中找到事务，因为前面的示例场景在这一领域极其常见。你会在任何需要“非此即彼”内部锁的系统中发现该主题的变种。

失败注入

到现在为止，我们已经讨论了代码如何应对可能的失败。从测试角度来讲，你如何确保代码会在关键资源无响应、死亡、用尽、失效、终止和成为过期资源时正确响应呢？

解决办法是使用自动测试用具注入失败。这用模拟对象框架最容易办到，因为你可以告诉模拟对象返回正常数据几次，然后返回可疑数据或抛出异常。跟普通测试一样，你在测试代码中验证恰当的异常已经抛出。

回顾列表更新问题，这里的测试代码为关键字 1 模拟有效的数据库响应，为关键字 2 模拟查询错误：

ListUpdate2.rb

```
require 'rubygems'
require 'test/unit'
require 'mocha'

class ListUpdateTest < Test::Unit::TestCase
  def test_database_failure
    database = mock()
    database.expects(:query).with(1, :name).
      returns('Anand')
    database.expects(:query).with(1, :address).
      returns('')
    database.expects(:query).with(2, :name).
①     raises
    q = ShippingQueue.new(database)
    q.add(1)

    assert_raise(RuntimeError) do
②     q.add(2)
    end

    # 列表仍然完好
③   assert_equal 'Anand', q.head.name
```

```
      assert_equal nil, q.head.next
    end
  end
```

① 注入 RuntimeError 异常。

② 调用将抛出异常，assert_raise 正期望得到它（并捕获这个异常）。

③ 验证列表仍然完好，仿佛 q.add(2) 从未调用过。

这类失败注入让你可以全面考虑——并验证——每个潜在的崩溃场景。用这种方法进行测试跟你测试正常路径一样常见。

测试猴子

你可以花上一整天仔细考虑各种场景，构建出非常健壮的代码。可大多数防错程序还是会被足够聪明的傻瓜给挫败。如果你手头没有这样的傻瓜，接下来最好的工具就是测试猴子（test monkey）。

在我第一份从事手持计算机的工作中，有一个称为猴子（monkey）的程序，它会向 UI 层注入随机的触击和拖拽，就好似来自触摸屏的操作一般。没有比这更绝的事情了。我们会一直运行猴子，直到系统崩溃。

猴子可能不是一个聪明的傻瓜，但一大批像疯子一样、24 小时不停触击的猴子弥补了天才的缺失。天呐，没有莎士比亚（但可能有些是 E. E. Cumming ⊖），只有大量的崩溃。这些崩溃是我们没有预料到的——这才是关键。

你能用相同方法做出一个用随机（但有效的）数据使你的程序几乎崩溃的测试用具吗？让它循环运行上千或上百万次，你永远不知道会发生什么。我在最近的项目里用到了这个技术，发现一旦在第二个满月，某供应商的 API 函数就会对虚拟机状态返回“未知”。它们想表达什么，它们不知道这个状态？我完全没想到函数会返回这个值。我的程序会在它出现时崩溃。我又一次接受了教训。

⊖　E. E. Cumming（肯明斯），美国现代诗人（1894—1962）。——译者注

行动指南

回顾前面客户列表的代码。你该如何修复它？这里是一段跟它一起工作的代码：

ListUpdate2.rb

```
require 'thread'

class Customer
  attr_accessor :name, :address, :next

  def initialize
    @name = nil
    @address = nil
    @next = nil
  end
end

class ShippingQueue
  attr_reader :head

  def initialize(database)
    @database = database
    @head = nil
    @mutex = Mutex.new
  end

  def add(customer_id)
    # 填写这部分
  end
end
```

用前面失败注入部分的代码来检查你是否正确。

技巧 6

确定风格

[白带] 在职业世界面前，书写风格优秀的代码将带来极大帮助。

下列两个函数的功能完全一样：

Fibonacci.c

```
uint64_t
fibonacci(unsigned int n)
{
    if (n == 0 || n == 1) {
        return n;
    }
    else {
        uint64_t previous = 0;
        uint64_t current = 1;

        while (--n > 0) {
            uint64_t sum = previous + current;
            previous = current;
            current = sum;
        }

        return current;
    }
}
```

Fibonacci.c

```
unsigned long long fbncci(unsigned int quux) { if
(quux == 0 || quux == 1) { return quux; } else {
unsigned long long foo = 0; unsigned long long bar
= 1; while (--quux > 0) { unsigned long long baz =
foo + bar; foo = bar; bar = baz; } return bar; } }
```

你愿意维护哪一个？

这个例子有点极端，但它阐述了一个简单的观点：你的代码不只给编译器读，它也供其他程序员阅读。书写风格优秀的代码是软件质量的一个要素，因为你根本无法维护你读不懂的代码。

风格要素

广义的风格指的是编译器不关心但人类关心的所有事情。这里有些例子：

- 类、方法、变量、字段命名等
- 单文件和多文件内的函数分布
- 注释
- 大小括号（哪里可选）
- （等价）控制结构的选择
- 大小写
- 缩进和其他空白符

风格优秀的定义不是一成不变的，这取决于你合作的程序员、项目或公司风格指南，以及编程语言建立起来的惯例。但共性还是有的，我们将在后面看到。

命名的重要性

良好书写的代码读起来不会像人类语言，但也不应该读起来像古怪的象形文字。类、方法、参数和变量的优秀命名将大大有助于代码让另一位程序员读起来自然。这并不意味着名字需要过于冗长，它们只需要适合问题域就行了。

考虑本技巧一开始的 Fibonacci 代码。变量 previous、current 和 sum 都体现了它们的用途。参数 n 有点短，但对问题域是合适的；函数的用途就是返回第 n 个斐波纳契数。类似，i 和 j 常用作循环变量。

若你正纠结于如何给某个事物命名，那就暗示着：代码的目的可能存在问题。这里是个例子：

```
im = InfoManager.new
puts im.get_customer_name_and_zip_code(customer_id)
```

InfoManager 到底是什么？用它来干什么？你如何推断它？像 InfoManager 这样模糊的名字往往标志着模糊的用途。方法名同样暗示出了有问题的代码。对比以下代码：

```
customer = Customer.find(customer_id)
puts customer.name
puts customer.address.zip_code
```

像 customer 和 address 这样的对象是你可以推断出的事物，并且听起来自然的方法名——find()、name() 等——应该很自然地表明了其意图。

注释

传说中终极糟糕的代码注释当然要数：

```
/* 给 i 加 1 */
```

注释不应该告诉读者代码是如何工作的，这是代码的职责。如果代码不清晰，那就把它改到清晰为止。取而代之，把注释的重点放在以下事项上：

- 若代码不直观，那它的目的是什么？例如，IMAP 协议将用户的收件箱定义为特殊的字符串 INBOX，因此，代码中的注释会告诉读者去参考规范中的相应小节：`list("INBOX"); /* INBOX 邮箱是特殊的，参见 RFC3501 小节 5.1 */`。
- 期望的参数和返回值是什么？有些可以从参数名推断出来，但对于公共 API 而言，函数前的一段概括性注释会很有帮助。同样，很多文档生成器会扫描源文件，生成公共 API 的概述文档。JavaDoc ⊖和 Doxygen ⊜是这类任务的常用工具。
- 有没有什么是你需要临时记下的？程序员会使用 TODO 和 FIXME 这样的字符串在开发中提醒自己。但要在检入前修正这些字符串：如果真的要在以后做某件事，把它记录在团队使用的任务跟踪系统中。如果它是一个 Bug，修复它或记录到 Bug 报告中。源代码不是你的待办事宜列表或 Bug 数据库。
- 文件的版权和许可证是什么？常用的实践是在每个文件放入一个注释头，指明版权所有关系（一般是你的公司）和任何许可证术语。若拿不准，那就没有许可证，即“保留所有权利”。贡献给开源项目的代码需要明确指出许可证。

若使用正确，注释会以一种自然的方式对代码进行补充，给未来读者一幅清晰图画，帮

⊖ http://java.sun.com/j2se/javadoc/
⊜ http://www.stack.nl/~dimitri/doxygen/

助他们了解发生的事情和原因。

退出和异常的惯例

这是局部风格，具有局部正确性。一些风格指南，一般针对 C 代码，指出函数只能有一个退出点。这条规则的本意往往是想确保任何分配的资源都释放了。我在几个操作系统内核中曾看到过类似这样的代码：

ExitPoints.c

```
int
function()
{
    int err = 0;
    char *str = malloc(sizeof(char) * 5);

    if (str == NULL) {
        err = ENOMEM;
        goto ERROR;
    }

    // ...

    FILE *file = fopen("/tmp/foo", "w");

    if (file == NULL) {
        err = EIO;
        goto ERROR_FREE_STR;
    }

    // ...

    err = write_stuff(file);

    if (err != 0) {
        err = EIO;
        goto ERROR_CLOSE_FILE;
    }
    // ...

ERROR_CLOSE_FILE:
    fclose(file);
ERROR_FREE_STR:
    free(str);
ERROR:
    return err;
}
```

在正常路径，执行将经过底部的 fclose() 和 free()，按与创建相反的顺序释放资源。在末尾使用标签能够让错误情况简单地设置期望的返回值，并跳转到释放相应的资源的位置。这在概念上类似抛出一个异常，只不过是你自己来调用“析构函数”。这种技术比手动检查每条返回语句要难犯错。

当然，其他 C 风格指南坚称永远不要用 goto 语句，违规者斩。若一家公司的风格指南同时坚持单一退出点和无 goto 语句，那请练好基本功来同时满足这两条规则。

当调用的 API（如 C 语言库）不提供含构造函数和析构函数的类时，异常可以采用类似的策略。可完成一个包装恰当资源的轻量级类往往会更好。这里是 C++ 的例子：

OpenFile.cpp

```
class open_file
{
  public:
    open_file(const char *name, const char *mode) {
        _file = fopen(name, mode);

        // 若为 null，则抛出异常
    }

    ~open_file() {
        fclose(_file);
    }

    //转换操作符，这样实例可被用作 fprintf 的参数
    operator FILE*() {
        return _file;
    }

  private:
    FILE* _file;
};
```

在本例中，open_file 实例可以在栈上创建，文件将在函数返回时关闭，不论正常返回还是异常离开——C++ 都将调用栈上任何实例的析构函数。

如有疑问

若你的公司没有编码风格指南，可以求助于以下内容：

- 编辑任何代码时，风格保持一致。比起蹩脚的风格而言，更让人烦的是混杂着多种风

格的文件。

- 遵循任何公认的语言惯例。某些语言，如 Ruby，对于命名和缩进都有非常好的先例可循。写 Ruby 程序时，要有 Ruby 开发者的样子。
- 若语言有不一致的先例，如 C++，遵循你正在用的主要库的先例。C++ 标准模板库有一致的命名风格，因此在使用 STL 时，遵照它们的风格是合情合理的。

对于使用多种语言的项目，遵循每种语言的惯例仍然说得通——让 Ruby 像 Ruby，让 C++ 像 C++。这远远超越了像命名和缩进这类问题，也要遵循每种语言的习语。参见第 2 章的“用习语编程”一节了解进一步的信息。若需要，提供一个桥接层。

进阶阅读

读读 Robert C. Martin 的《Clean Code》（代码整洁之道）[Mar08]，它是编码风格的权威著作。参考 Wikipedia ⊖可获得大量风格指南的链接。

行动指南

找到你使用语言的风格指南（有时称为编码标准），优先考虑对每条规则的理由进行了解释的那份指南。有些规则有点随意，但大多数的目的是为了减少偶发 Bug 或改善可读性。了解规则背后的原因，而不是仅仅知道规则是什么。

⊖ http://en.wikipedia.org/wiki/Programming_style

技巧 7

改善遗留代码

[白带] 维护和改善遗留代码是从入职第一天起就要面对的现实。

假若能简单地让所有废旧代码消失，把它们扔进垃圾箱，然后重新开始，你的工作会（看起来）轻松得多。但这不会发生，那你要怎么做？

典型的怪兽遗留代码库有如下特点：

- 函数跨越数千行，有近乎无限的可能代码路径。
- 类或模块依赖 20 个（或更多）其他类。
- 某处有这样的注释：“别动这里，否则系统将停止工作！”
- 另一个注释写道：“该代码请问问 Bob。”这个 Bob 是 10 年前离开公司的程序员。
- 多得说不完。

有时，当需要修复代码里类似这样的 Bug 时，阻力最小的路径——恰好不用做任何整理工作就可完成修改——是最谨慎的路径。但俗话说得好：“不要把事情越描越黑。”假如需要花段时间维护这段代码，最好就是“边维护，边改进”。

找出接合点

清理遗留代码的关键问题是：从哪儿开始。若所有代码都彼此依赖，如何分离出一个可工作的模块？假设你正在遗留的 Win32 应用程序上工作，准备将它迁移到 POSIX。系统 API 是一个好的起点。或许可以从文件 I/O 着手，寻找类似下面的代码：

行业观点：向遗留代码进军

开始在遗留项目里工作时，挑件非常小的事情，完成非常小的改动，然后观察其影响。要是遗留代码库有完善的测试套件将会更好，但可能指望不上。更糟的是，测试套件的设计方式可能几乎无法完成对改动的测试。

增加测试用例可能会很困难。代码彼此纠缠，紧密耦合，即使能从中剥离一小块进行测试，也只能投入相对简单的测试。真正困难的部分让测试非常难以进行。

这是这场战役中最难的部分。你必须找到能插入你的进步之旗的地方，编写可以理智和清晰地控制系统那部分行为和英勇保卫它的测试。一旦完成一次侵入，就寻找可以壮大胜利成果的方向，顽强地继续追击。

——Rich Rector，Spectra Logic 工程经理

```
HANDLE hFile;
if (CreateFile(hFile, GENERIC_READ, 0, 0,
               OPEN_EXISTING, 0, 0)) ==
               INVALID_HANDLE_VALUE) {
    // ……错误处理……
}
```

与其用 100 个 POSIX 调用替换 100 个 Win32 API 调用，不如借机会将文件 I/O 抽取成自己的模块。（或使用像 Apache Portable Runtime 这样现成的跨平台库[⊖]）同时实现这个模块的 Win32 版和 POSIX 版，因为这能让你验证程序在两个平台上的行为。

这种抽取出功能点的实践有时称为找出接合点（finding the seam），因为你在寻找可以将遗留代码拆分的自然位置。尽管一开始没有太多的接合点，但情况会越来越好。每个新构建的模块都是模块化、经过良好测试的，因此，当抽取下一级接合点的时机到来时，它给你提供了一张更大的安全网。

⊖ http://apr.apache.org/

迁移到新平台和新语言

计算世界从未保持过平静，遗留系统有时需要迁移，同时保持功能不变。它可能是由 Windows 的某个古老版本迁移到当前版本；在野心更大的项目中，可能是将系统由 PC 转变到 Web。

只要可能，就通过复用部分旧程序遏制移植风险。这里有些例子：

- 若旧程序是用像 C 这样常见的语言写的，许多其他编程语言都有一种跟 C 代码打交道的方法（如 Java 本地接口、Ruby 扩展等）。
- 若旧程序有网络或命令行接口，你可以通过屏幕抓取技术构建一个跟旧程序交互的填充层。你可能觉得可笑，但为古董级大型主机系统构建新型前端时，这是常用方法。

这些或许不是创建可维护系统的最佳解决方案，但它们可能为你节约时间。考虑另一个场景：公司的遗留系统建立在有上千个已知安全漏洞的 Windows 某个版本基础上，每个人都生怕现在迁移系统，他们愿意尽可能省事。采用一种折中步骤——同时还能给团队赢得时间，让他们把事情做好——刹那间，听上去也不坏。

Bug 与不良特性

程序员新人的常见任务是 Bug 巡检。祝你好运。修复遗留代码里的 Bug 时，要小心区分 Bug（明显的错误行为）和看上去奇怪的代码。修复陌生的事物会以你意想不到的方式“咬你一口”。

假设你正工作在一个 Web 浏览器上，当它想产生某个 HTTP 标头的字段时，它会崩溃。这听起来像是个明显待修复的 Bug。可在修复那个 Bug 时，你还注意到浏览器创建了一个标记为“Referer,”的 HTTP 标头，它拼错了。要不要修复它？

对这个例子的答案是“不”。许多 Web 服务器都依赖于这种拼写错误——事实上，这要追溯到 20 世纪 90 年代中期出的 RFC 1945。“修复”这个标头将破坏所有部分。

这里并不是说你不该修复陌生的东西。但要意识到奇怪的代码可能是有原因的。问

问你的师傅或资深程序员。至少在检入注释里写上你的变更，万一 Bug 其实是不良特性（misfeature）伪装时，其他人可以快速地找到它。

进阶阅读

大多数编程书籍关注的是如何写新代码。你不能因此而责怪作者或购买这些书的程序员；“绿场”（green-field）[⊖]编程肯定快乐多多。有两本书是专门针对“灰场”（brown field）编程的。

Michael Feathers 的《Working Effectively with Legacy Code》（修改代码的艺术）[Fea04] 是对付遗留代码的权威书籍。假如你正在一个大型遗留项目上工作，这本书就是为你而写的。

在更加战术性的层面上，Martin Fowler 的《Refactoring: Improving the Design of Existing Code》（重构：改善既有代码的设计）[FBBO99] 对于任何维护上年头的代码的人都会有帮助。

行动指南

有些开源项目历史悠久，可它们并没有像传统遗留代码那样退化成乱麻一团。看看 Apache HTTP Server [⊜]，首次发布是 1995 年；或者是 FreeBSD [⊝]，于 1993 年第一次发布。就在撰写本书的时候，它们两个都还处于活跃的开发状态。

这两个项目的标志是它们清晰的代码库。假如有点关于 C 的知识，你可以随机挑个文件，毫不费劲地理解代码的作用。按以下步骤执行：

- 下载这两个项目中任意一个的源代码，或者使用它们的在线源代码浏览器阅读代码。
- 观察它们对于单一编码风格的执著，以及这种风格能让你轻易地阅读好几页源代码。

⊖ “绿场”和“灰场”这两个术语出自建筑行业。“绿场”指的是未开发（尤其指未污染）的土地；“灰场”则与“绿场”相反，指的是开发过（通常是污染和废弃）的土地。——译者注

⊜ http://projects.apache.org/projects/http_server.html

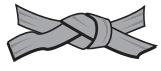
⊝ http://www.freebsd.org/

- 注意它们是如何将公共模式抽象成单独库的，如 Apache Portable Runtime ⊖，它让核心代码更容易理解。
- 思考：这些项目可能有点老，但一点也不像遗留项目，几乎没有理由用某个更新的库来替代它们。它们是怎样做到与时俱进的？
- 思考：你能将这些项目使用的编程技术或标准用到自己的公司里吗？

⊖ http://apr.apache.org/

技巧 8

代码审核要早且多



[褐带] 你的代码或许不会在第一天就被同行评审，但这种事情头几个月内肯定会发生。

许多程序员厌恶、憎恨或双倍不喜欢代码评审，但实在没有理由去恨它们。其实，有经验的程序员还期待代码审核——我们很快将看到原因。

人和观点

如此多的代码审核以失败告终的原因在于：程序员将他们代码的价值和自身的价值画上等号。当审核者指出代码中的问题时，程序员将之视为一种攻击，并且采取防卫姿态，整件事情于是很快急转而下。

我们要清楚地认识到这一点：审核者将找出你代码中的错误。我敢保证，绝对会找到。但那并不意味着你的技术就很糟糕。改进的空间总是存在；或者对于你的代码该如何书写，至少存在不同观点。将代码评审核为一种开放式讨论，而不是一种你是被告的审判。

“错误”的范围可以从 Bug 到风格问题。Bug 简单，你必须修复它们。每个人，菜鸟和专家都一样，不时会犯错，因此屋子里没人会认为你就是白痴。把问题记下来，然后继续前进。

更具有争议性的问题在面对风格问题时产生。可能你正在使用带计数器的循环，资深程序员审核你的工作时建议换成迭代器。这表明你的代码有错和该建议正确吗？不，风格问题没有那么绝对。

既然这是一场开放式的讨论，那就可以深入探讨这个建议的优点。审核者可能会说：“使用迭代器消除了差一（off-by-one）错误[⊖]的可能性。” 不要产生防御心理并争辩：“可我的循

⊖ 即在边界情况时少一或多一。——译者注

环没有差一问题！”审核者已经看到了。他想说的是：换种方法消除那种可能性是好风格。

一旦理解了审核者的意思，就要感谢他的建议，记下来，继续前行。在你有时间让代码审核的紧张感消失之后，考虑你的行动计划。风格上的异议会因为人身攻击引发争议；若在没有他人在场的情况下考虑这种异议的优点，你可能会发现审核者有一定的道理。

观点是本质：无关乎你和审核者的对错。它关乎好的代码和更好的代码。

形式

我将描述我看到过的代码审核的形式，并就它们给你一些提示。

非正式审核

你经常会被某些东西难住，这时需要有人来帮你解决它。或者可能是，你发现这是个好的解决方案，但你拿不准。去找个更有经验的程序员。就算是脾气再不好的人这时也往往会先忍住，被征求意见的恭维甚至能软化脾气最差的人。

伙伴系统

有些项目要求凡是要检入代码库的代码都要有“伙伴”签字。你完成了变更，也测试过了。现在，你需要有人在检入前对其评审。不要只找跟你要好的同事，他们会对你写的任何东西大开绿灯。去找你更改部分的领域专家。要是找不到，那就去找一个跟你不太熟的同事。

将伙伴系统作为一种让更多人熟悉你工作的方法。尤其当你是新人的时候，没有什么比用代码来建立自己的公信力更好的方式了。它不需要是耀眼、巧思、别出心裁的代码——只需要是可靠的代码。确保人们可以定期地见到它。

高层审核

这往往是多人参加的有投影仪的正式审核会。往往审核数周的工作成果，但以一种高层次的观点来看。你解释设计，解释它如何转换到代码，然后审核代码的关键部分。这是一次讨论设计和风格的机会。准备好挨批评，记住在前面的“人和观点”一节讨论过的问题。

作为审核员，我最喜欢的问题是“给我看测试”。我对我领导的每个项目都要求自动化测试，因此，要是对这个问题的反应是一片茫然，审核就结束了，我们将在下周安排一次新的审核。不管怎样，有经验的程序员会从浏览测试开始审核。没有什么比展示测试更能确立对你代码的信心了。

逐行审核

最无聊、让人觉得压抑的代码审核就是每人逐行检查代码。实际上，这类审核往往是为那些已经成为灾难的代码准备的。（最好不是你的代码。）假设你处于捉虫模式，对每行代码都要问：这行的假设是什么？哪些方法会让它失败？万一失败了会怎样？

如何避免这种对你代码的逐行审核？简单：让你的代码早点审核，而且经常审核。采用非正式或伙伴系统，或安排组内审核。我一开始说过：有经验的程序员渴望代码审核。这是因为他们更愿意早点得到反馈，避免陷入需要逐行审核的麻烦里。

审查

我从别人那里听到这个实践，但自己从未用过。在审查过程中，资深程序员审视你的整个项目，深入到具体主题。我说的“深入”其实是顺藤摸瓜的意思。为何你要选择某某设计？你有哪些数据证明你的假设？如何证明（或测试）你的实现是正确的？要是它能扔木头，它每秒可以扔多少木头？明白了吧。

准备审查是件大事，因为你不知道审查员会问什么。你必须准备任何事情。我在这里唯一的建议就是：在编程时，问自己同类问题。要是你的程序是从文件中读取数据，问问自己，我对文件格式做了什么假设？我是如何测试这些假设的？文件可以有多大？我如何证明？

当然，你目前可以只采用一种方式。但这种思维方式的回报总会在某个时间点开始递减；这个时间点取决于项目的类型和它的生命周期阶段。对于产品代码，小心谨慎为佳；对于商展示例或概念证明，只需完事就万事大吉了。

代码审核策略

关于代码审核策略的范围从不存在直到有制度保证。若团队的开发风格处于极度混乱的状态，除非迫不得已，他们不会去做代码审核。那时，等待你的将是侵蚀生命的逐行审核。若团队采用像极限编程这样的开发实践，代码审核就是常态：XP 的结对编程，在效果上就是边写代码边审核。

有的行业出于认证的目的要求代码审核。如果你正在为航空电子或核电站写软件，在发行软件之前，会有一个令人发抖的审核过程。你知道有好多软件都附带一个最终用户许可证协议，上面基本上写的都是软件没有保修期且随时有可能崩溃吗？在航空电子业可没有这种好事——你的代码可是人命关天的。

然而，对于我们中的其他人，并不存在唯一正确的审核方法。最好的代码审核策略就是能带来最大好处的那个，依不同的团队和项目而不同。（提示：答案永远不是“从不审核”。）有经验的经理或技术领导应该根据需要设定策略。

不管怎样，你总是可以召集审核。当想让他人看你的代码时，只管去找，不必害羞。有经验的程序员总是这样做。若是初级程序员不请求审核，那肯定是麻烦滋生的信号。

行动指南

主动要求下次代码审核：在准备将下一次变更集检入团队的代码库之前，找位同事审核你的代码。但在找他之前，得做点功课：

1. 产生改动的文件列表。源代码控制系统应该可以方便地告诉你这些信息；一般是“status”命令。

2. 收集每个文件的差异，最好用图形化的对比工具，它会让你同时看到原始副本和当前副本，变动部分会突出显示。

3. 现在，不要省去这一步，自己整体看看变更，确保你可以解释它们每一个。我不是指的是像审查风格那样，逐行深入了解你的动机，而是寻找明显的疏忽。里面很可能会有无意

间引入的错误，把它改过来，重新获取新的差异。

现在，找个伙伴。要是你的团队没有明令这样做，只要找另一位程序员（最好是比你资深的人）就行，像这样说：“你愿意在我把代码检入之前看一下我的改动吗？”

拉上你的伙伴，在屏幕上展示差异，解释你修改的目的，然后对每个文件的差异逐个审阅。可以是你或伙伴来主导，只要效果好，哪种都行。假如你的编程风格很优雅（为什么不这样要求自己呢），伙伴应该可以很快看完你的变更。

除了解释代码外，还要解释你的测试方法。理想情况下，你会有一个自动化测试套件；也审核这些代码。要是没有，说明一下你做过的任何手动测试或证明代码的正确性。

甚至在叫伙伴过来之前，你也很可能会找到一两个疏忽。此外，你的伙伴会问一两个你没有想过的问题。你可能会发现，每次提交时都需要一个检入伙伴。

New Programmer's Survival Manual
Navigate Your Workplace, Cube Farm, or Startup

我喜欢这本书中务实的论调和内容。

—— **Bob Martin** Object Mentor公司总裁，著有《The Clean Coder》

这本书极好地总结和叙述了软件开发新手缺失的软件开发“宏观视图”，以及许多其他的方面。这是一本开启全新软件职业生涯的优秀入门书。

—— **Andy Keffalas** 软件工程师和团队领导

这本书阐述了不断发展和变化的IT行业潜规则，观点新颖有趣、内容深入浅出。假如你刚拿到计算机学位，一定要读一读这本书。

—— **Sam Rose** 格拉摩根大学计算机科学系学生

这本书包含了所有在我刚入行时就应该学习的知识，是软件开发新手的必读书籍，对行业里的所有人来说，同样开卷有益。

—— **Chad Dumler-Montplaisir** 软件开发者

无论从事哪个行业，从门外汉到专家，都不是一件容易的事，需要付出很多努力，并要走很长一段路，软件开发这个行业也不例外。假设每一位程序员的能力和付出的努力都是一样的，但最终取得的成就和所花费的时间也可能完全不一样。如果你能遇到一位经验丰富、循循善诱的良师，也许你就能走捷径；如果你独自去摸索，则很有可能走弯路。每一位致力于从事软件开发工作的程序员都希望能在自己专注的领域成为专家，而且希望时间成本和试错成本尽可能低！本书作者Josh Carter就是这样一位具有20年丰富编程经验的导师，他非常清楚软件编程新手需要掌握哪些技能和素质，并对他们在成长过程中可能会碰到的各种问题了然于胸。Carter将他的这些见识和经验系统性地总结成了这本书并奉献给所有致力于成为专家的程序员新手。

本书从编程技能和方法、编程工具、自我管理、团队协作、组织架构、工作态度和原则、自我学习和持续改善等多个方面系统总结和阐述了专业程序员应该掌握的33个技巧。如果能认真研习本书，你便踏上了一条通往专业程序员道路的捷径！

上架指导：计算机/程序设计
ISBN 978-7-111-41164-2
9 787111 411642 >
定价：49.00元

The Pragmatic Programmers

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com